

# A Visual Shell Scripting Tool

*Eugene Tseytlin<sup>1</sup> and Shi-Kuo Chang<sup>2</sup>*

Department of Computer Science  
University of Pittsburgh, Pittsburgh, PA, USA  
tseytlin@pitt.edu<sup>1</sup> and chang@cs.pitt.edu<sup>2</sup>

## **Abstract**

*This paper presents a visual shell-scripting tool that enables creation of Unix shell scripts from individual components that wrap various Unix programs. A usability study was conducted to compare programming using VisualDesktop with traditional shell script programming. Extensions to this tool to include software patterns, or templates, for both experienced and novice programmers are suggested.*

**Index terms** — Visual programming, visual software engineering, patterns, templates.

## **1. Introduction**

Visual programming languages have their appeal, as they are easy to learn, easy to visualize, and easy to write. Historically, in most applications, true visual programming languages were used only as a learning tool for children or handicapped people. Experienced programmers traditionally prefer conventional languages, as they are much more flexible and powerful [5]. Programmer is usually able to design the program much faster in conventional language, than by trying to build it visually from UML or flowchart constructs. Exception to that are the hybrid languages such as Microsoft Visual Basic/C++ or Borland JBuilder, where only the graphical user interfaces (GUI) are built visually. There is one area where the true visual programming language would be useful for any programmer or advanced user: Unix shell scripting.

UNIX is an interactive time-sharing operating system invented in 1969 by Ken Thompson after Bell Labs left the Multics project, originally so he could play games on his scavenged PDP-7. Dennis Ritchie, the inventor of C, is considered a co-author of the system. By 1991, Unix had become the most widely used multi-user general-purpose operating system in the world. Many people consider this the

most important victory yet of hackerdom over industry opposition. [1]

Unix is a very popular and comprehensive operating system. It is simple and elegant: one of its most powerful features is the ability to save the set of commands in a file, called a shell script. The philosophy of Unix is to use a large set of small simple programs to accomplish complicated tasks. Those programs are being communicated via pipes and/or file redirections and can be used in a batch mode. The problem is that a majority of those programs have a very extensive and often-non-intuitive command line options. The material is completely reference oriented with very little tutorial information and beginning users often find it overwhelming. That means that whenever a programmer or an advanced Unix user decides to write a shell script, he or she has to use help system to look up a small utility command. This can become a very time consuming and tedious task.

This task can be greatly simplified if those utilities had a graphical front-end. However, it is not enough to create a front-end for every single Unix utility to make them easier to use. The new system that would allow for those utilities to be used together needs to be developed and explored. The system would also allow programs to be piped into each other, redirected to a file, or be inside the for-loop or an if-statement. The system presented in this paper will attempt to do just that.

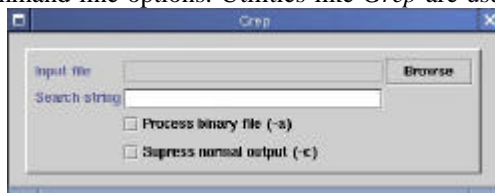
It was inspired by a fiswidgets project developed by Kate Fissell (University of Pittsburgh) and Tim Magela (Psychological Software Tools). “Fiswidgets (Functional Imaging Software Widgets) is a set of Java classes built on top of the Java AWT and Java Foundation Class (JFC, Swing) toolkits; it includes both Graphical User Interface (GUI) components and thread control components and is designed to permit a programmer to very quickly write simple Java graphical user interfaces to run application programs that would otherwise be invoked by a command line or from within a shell script.” [2] One of the main

appeals of fiswidgets is the use of meta-fiswidgets the Desktop in particular. Meta-fiswidgets allow users to run a set of applications called “flow” one after the other. Fiswidgets are currently used to wrap complex neuro-imaging applications, however this framework can be extended to other domains including Unix shell scripting. With additional programming constructs such as variable definitions, loops, and conditionals, creation of fully featured Visual Shell Scripting Tool may be possible.

## 2. System Design

A subset of standard Unix utilities is wrapped with Java GUI(s). The user interface will easily allow users to run most of the Unix programs. The same interfaces will be used from within a tool called VisualDesktop, where iconic representations of each application will be arranged in the form of the Unix shell script.

Below is an example of a stand-alone *Grep* Unix utility wrapped in the Java GUI. Text fields, check boxes and browse buttons replace tedious command line options. Utilities like *Grep* are usually



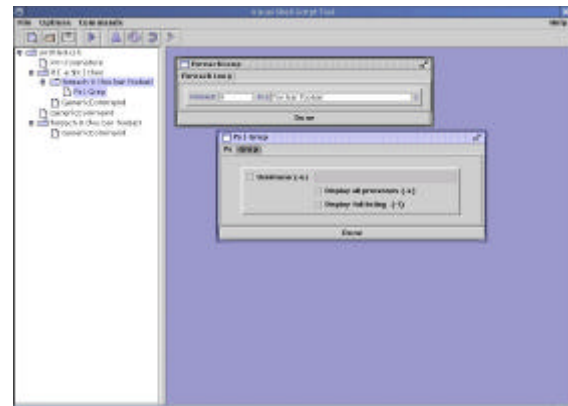
utilized in conjunction with other Unix programs. VisualDesktop allows users to run several applications in sequence or pipe the output of one program into the other. Representation of program flow will be illustrated visually in the VisualDesktop.

VisualDesktop is a meta-application. It enables the creation of a shell script from individual components that wrap various Unix programs. Components are added to the flow when their names are selected from a pull-down menu. Along with components the programming constructs such as *if statements*, *variable definitions*, and *foreach loops* can be added to the flow.

VisualDesktop consists of two major parts: the *source tree* on the left hand side, and the *desktop panel* on the right. The *source tree* is an iconic representation of the shell script that is being created, where each node in a tree represents a component or a set of components that were added to it. The *desktop panel* is the area where the interface for each component is displayed.

The figure below is a screenshot of VisualDesktop. The white panel on the left is the source tree of the currently loaded shell script. The

larger blue panel on the right is a workspace, where properties of individual components can be set and modified.



## 3. Major Elements of Visual Desktop

### 3.1. Source tree

The *source tree* is an iconic representation of a shell script that is being created. The tree data structure has been chosen for its familiarity and its resemblance to the properly indented code in the procedural language such as Unix shell script.

All of the elements are children of the root node that represents the shell script itself. Elements that represent programming constructs, such










as conditionals (*if statements*) and loops (*foreach loops*), are also parents to elements that are in their *code block*. Code block is the set of statements that is going to be executed, when the condition in their parent is satisfied. The rest of the elements in the source tree are variable definition, Unix commands, and generic commands. Those elements are leaf nodes and cannot have children. Each element in the source tree is represented by two parts: icon image and short description. Icon image depends on the type of the programming construct that is being used, while description is the name of the individual application, set of applications, or a variable name. When the icon is double-clicked, the GUI of respective application will pop-up on the desktop panel.


The elements are added to the source tree in two ways. Variable definitions, conditional statements, *for* loops and generic commands are added when an appropriate button on the toolbar has been pressed. All of the wrapped Unix commands are added from an appropriate pull-down menu. Elements are added to the root node in the order they are selected. To arrange them in a desired order they must be dragged to an appropriate location or grouped with other icons. When the element is dragged its location is changed to the line it was dragged to. Grouping works by first selecting multiple elements with Ctl-Click or Shift-Click, then right clicking on the last element to bring up a pop-up menu and selecting *group* option. If one of the elements is the conditional or loop element, then the rest of the elements become a part of its code block. When a command is grouped with the other command, then it becomes a *pipeline* element, where the latter selected command will be piped into a former.

## Toolbar

Toolbar features most commonly used buttons, which give quick and easy access to frequently used functions of the VisualDesktop. All of these buttons are also duplicated in the main menu.

- |   |                    |  |
|---|--------------------|--|
|  | <b>New</b>         | Creates a new shell script.  |
|  | <b>Open</b>        | Opens saved shell script. Since VisualDesktop only implements a subset of <i>bash</i> , shell scripts created by hand will load, but will not be completely understood by a program.                       |
|  | <b>Save</b>        | Saves visual shell script. Visual script is saved as a regular Unix shell script that can be run in regular Unix environment.  |
|  | <b>Run</b>         | Runs the visual shell script. The source tree is parsed to create an actual Unix shell script and the shell script is executed as a separate thread. The output is displayed in the <i>output window</i> . |
|  | <b>Variable</b>    | Adds a <i>Variable Definition</i> element to the source tree. Variable definition will be explained later in the text.   |
|  | <b>Conditional</b> | Adds an <i>If statement</i> element to the source tree. Conditionals will be discussed later in the text.  |
|  |                    | Adds a <i>foreach loop</i> element to the source tree. Refer to the section on   |

**Foreach loop** source tree. Refer to the section on Loops later in the text.

 **Generic command** Adds a *generic command* element to the source tree. Generic commands will be discussed later.

## Commands menu

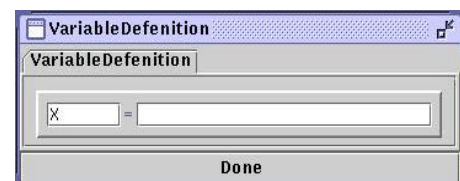
Menu in the VisualDesktop duplicates some functions that can be accessed from the toolbar. It also provides access to the set of Unix commands that were wrapped in a Java GUI(s). Each command in the *commands* menu is named after the program that it is wrapping. The names of Unix commands are usually not very descriptive, however knowing the names of the back-end programs used in the script would help the user navigate in the Unix environment. Each menu item also has a related *tooltip*, which gives a much better description of what that particular program is doing.



## 3.2. Variables

Support for variables is essential in any programming language. Variable definition is represented by an interface similar to the one used in other programs.

There are two fields in this interface: variable name,



and variable value. The syntax for the variable values is the same as for the conventional shell script. After the variable has been defined, it can be used anywhere in the script. To use the variable, one must simply put `$VARIABLE_NAME` in the appropriate field of any application.

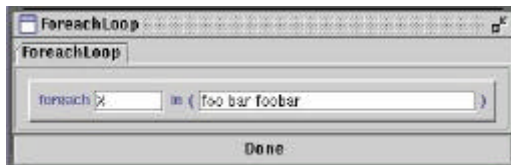
### 3.3. Conditional (if) statements

VisualDesktop supports the basic if statement. If expression inside the conditional statement is evaluated to *true*, then the code block that follows it, will be executed.



#### Foreach loop

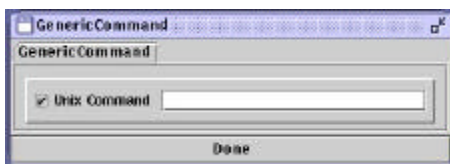
*Foreach* loop is represented by a panel that has a variable name on the left hand side. The list of



values, the variable will take with each iteration is located on the right hand side. Utilization of Unix commands will be very useful to generate that list. For example, *ls* command can be used to iterate over files in a specific directory. The number of iterations depends on the number of items in the list.

### 3.4. Generic commands

*Generic command* element represents a single Unix command that does not have an interface. Since there are hundreds of standard programs and command line utilities written for Unix, only a small subset of the most useful commands will be wrapped in the Java Interfaces. Programs that were not wrapped can still be used within a visual shell script by invoking generic command. Generic command window has a single field where the command has to be typed.

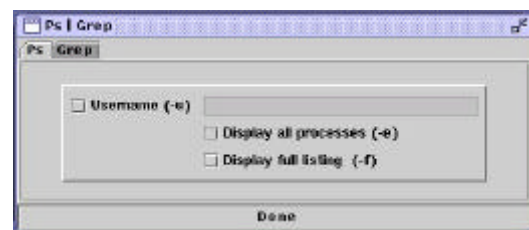


### 3.5. Pipes

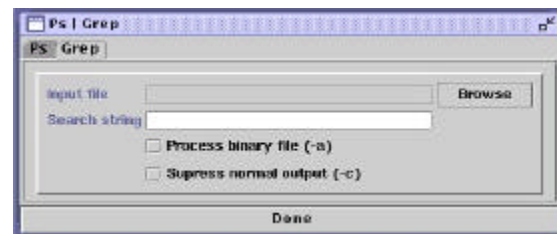
Pipes are very important in the Unix environment, especially inside shell scripts, because they allow several simple programs to accomplish

complicated tasks. VisualDesktop would have never become a usable visual shell-scripting tool, if pipes were left out of the design. In VisualDesktop pipes are represented by an icon with the names of commands concatenated with each other. When this icon is double-clicked, tabbed pane comes up. Each tab in the tabbed pane represents an individual program that will pipe its output to the next program in line. Example below represents two programs connected with a pipe: Ps and Grep. The icon that represents this command line is shown above, as well as a tabbed pane with showing both of the commands.

Screenshot below shows Ps Unix utility that is piping its output to Grep.



The next screenshot shows Grep utility that catches standard output from Ps and uses it as its input. Note that the Input field of Grep is disabled.



## 4. Usability Study

A usability study was conducted with the help of six volunteer programmers. Volunteers' backgrounds were very similar to those of the intended population of program users. All subjects were recent college graduates aged 22-26 with strong background in programming and Unix operating system, but neither one had any recent experience with shell script programming. The assignment consisted of three problems: simple, medium and hard. Each problem needed to be solved first by using VisualDesktop tool and then by using conventional means such as standard Unix editor. On-line help such as Unix man pages, Internet and several shell scripting books were available for both trials. [3, 4] To simplify the task, all three problems were given in pseudo code with some hints of which Unix utilities one might use for each task. All subjects were given a brief tutorial that explained basic functionality of

VisualDesktop. The first problem was relatively simple.

*List all processes that current user is running on local machine. Use PS utility piped into GREP to achieve that.*

The second problem was somewhat more complicated:

*Given a set of ASCII files in current working directory, generate a set of files that have some text string replaced with the other. FOR all files in current working directory use SED Unix command to find/replace an instance of text and redirect it to a copy of the source file.*

The third problem was relatively hard, as it involved several programming constructs, complicated command line and small number of lines of code.

*Given a set of ASCII files in some input directory, check that each file contains an instance of string A, B and/or C. If that is true, then insert a line of text at the first line of that file and save the result. FOR all files in a given directory and FOR strings A B and C, use GREP to see if the above string occurs in each file. IF GREP returned true, use SED to insert a string into file and redirect it to a copy of the source file.*

Four subjects participated in the initial experiment. Each subject solved every problem first visually and then non-visually. All problems were solved incrementally from easy to hard. If the subject could not solve some problem, the time and the outcome were recorded.

**Table 1.** Results from the initial set of trials (in minutes):

<i>Difficulty level of the task</i>					
<i>Easy</i>		<i>Medium</i>		<i>Hard</i>	
visual	non-visual	visual	non-visual	visual	non-visual
2	.75	5	10		
1	.5	7	10	60 *	
2	.17	15	16		
2	.17	5	10	60 *	25 *

\* Means that the task was not completed and user gave up after some time.

Results for the easy problem showed that simple one liner scripts are still easier to write in the editor or type in the command line. For medium

problem, shell script was constructed faster visually then non-visually. Hard problem trial was not successful. VisualDesktop was never tested with that kind of task and simply could not be used to produce a solution. None of the users could solve hard problem using visual or non-visual methods.

The reason why hard problem could not be solved were issues in the implementation of the VisualDesktop prototype. While great emphasis was placed on overall design, individual programming constructs and GUI wrappers for Unix command line programs were overlooked. None of the components had tool tips that were essential in understanding what each field does. Commands were inconsistent and some applications were incomplete. Application that caused all the grief was SED. Instead of having a proper user interface that could be easily manipulated, SED GUI had a single field that required a properly formatted command to be entered with only one tool tip as help. SED GUI implementation was not simplifying the command line and remained very reference oriented.

VisualDesktop had to be modified to solve this problem. Each programming construct had a useful tool-tips attached to each field. Some programming constructs like IF statement, had a reference widget implemented as a non-editable text area, which listed all available flags that can be used. New features were added, such as ability to use the output of executed commands as an input to ForeachLoop and VariableDefeniton constructs. More importantly, GUIs for individual applications were standardized and improved, especially for the SED application.

After modifying VisualDesktop the four subjects and two additional subjects were tested again. They were presented with same problems as before, but now they had a better working VisualDesktop. Since almost half a year has passed since previous trial, none of the subjects remembered solutions to any of the problems.

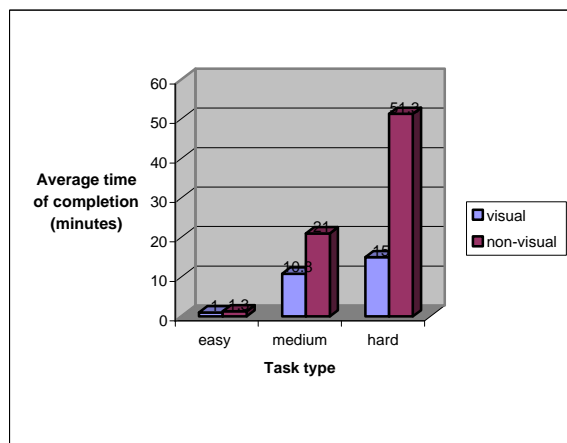
**Table 2.** Results from the second set of trials (in minutes):

<i>Difficulty level of the task</i>					
<i>Easy</i>		<i>Medium</i>		<i>Hard</i>	
visual	non-visual	visual	non-visual	visual	non-visual
1	.75	14	25	15	75*
1.5	1.5	10	20	25	29
1.33	1.25	12	19	13	51
.75	.5	14	34	17	65*
.5	.5	5	13	11	34
1	3	10	15	9	54



\* Means that the task was not completed and user gave up after given time.

The first four subjects in Table 1 and Table 2 are identical, and the last two subjects in Table 2 are the new ones. They were able to complete hard task successfully this time (at least visually). In fact, visual component of the medium and hard tasks took them roughly the same amount of time to construct. While easy and medium tasks took roughly the same time to complete as in previous trial, there was a great improvement in subjects' ability to get through the hard task. The averages of time it took users to complete each task are presented in the graph below.



While writing one-liner simple scripts takes roughly the same time to do visually as non-visually, significant improvement in performance on hard and medium task can hardly be ignored. It is interesting to point out the time difference between medium and hard tasks. While the difference between hard and medium tasks that were performed visually is only five minutes, the time difference for same tasks that were solved non-visually almost doubled. This indicates that once a user gets a hang of the VisualDesktop interface the performance significantly increases.

Correct Interpretation of results requires several clarifications. Since each problem was first solved visually, solving the same problem non-visually was a much simpler task. VisualDesktop gives a transparent view of the shell script program's design as well as some hints of what flags should be used for some of the Unix utilities. When problem was first solved visually, most of the design issues have been addressed. When participants started to write a shell script non-visually he/she already had a good idea of what the program layout should be and

what are the issues that need to be addressed. The only challenging part of the task remained to be shell script syntax as well as cryptic command line parameters for each Unix application. With that information in mind, one might say that results are slightly skewed in favor of non-visual approach, however the difference in time between two methods is still significant enough to show the efficiency of visual approach.

The experiment showed that the visual approach in designing a Unix shell script is faster than its conventional design. Not only such approach could be used as a learning tool for an inexperienced user, but it could also speed up development time for an advanced user by saving the trouble of looking up every other command in a reference book or a manual.

## 5. Patterns for Visual Shell Scripting Tool

Further improvement of the visual shell-scripting tool requires the introduction of templates or software patterns. Using templates or predefined code-blocks can significantly cut the development time by giving users ready-to-use components that can be quickly modified to fit the needs of a particular application. In the context of this paper, the terms "template" and "software pattern" are used interchangeably.

What is a software pattern? According to Alexander, pattern is a solution to the problem in certain context. [7] Patterns usually describe software abstractions used by programmers in their design. This approach helps programmers to solve reoccurring problems with relative ease. Whenever software designer encounters a problem, he or she can look-up a pattern that solves a similar problem and tailor the example solution to his or her needs. [6]

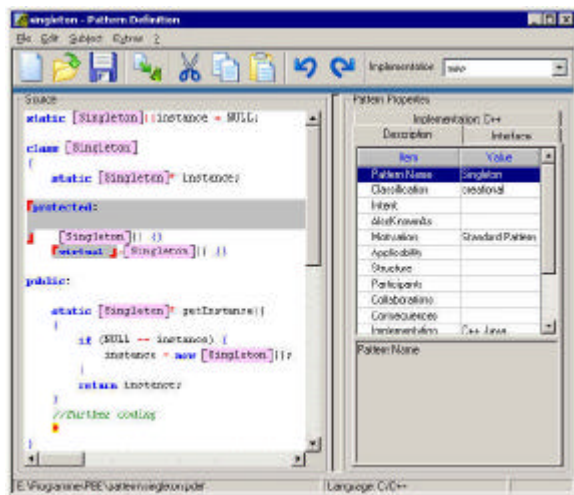
Unix shell scripting is not a fully-fledged programming language, but it can still benefit from traditional software design patterns. For that to be true, those concepts have to be geared toward Unix shell scripting environment. Presenting object-oriented patterns for procedural language such as Unix shell would not be reasonable.

In order to use software patterns, they need to have a common representation scheme. A general approach in books on *Design Patterns* is to use plain English text to describe the problem, context, and the solution. One can also utilize diagrams or some other method of graphical representations. Both of these methods cannot be integrated into VisualDesktop to assist the user at hand. They can be only available as a supplement. There is another way to represent

patterns that can be used within VisualDesktop. This method presents a pattern through the example.

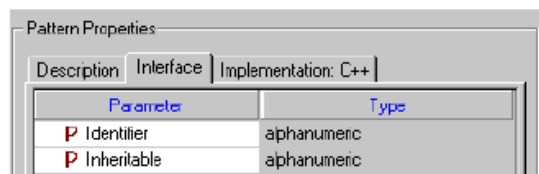
Example is a natural way in explaining not just patterns, but just about anything in our everyday lives. Example is usually included with any good pattern definition. It might be difficult to determine what the problem is or which pattern it represents just by looking at the example. Still, example will let users see the solution to the similar problem and from that most people can determine the pattern right away.

*Pattern by Example* is a commercial tool developed by Delta Software technology. This tool uses an example file written in one of the supported languages to build an intermediate pattern representation that can later be used to generate the source code for a specific problem [8]. In the figure below, the example source code is analyzed by the user. Variable sections of the code are marked as template parameters. All pattern properties including parameters are set and modified on the right panel.



*Pattern by Example* allows users to take out non-essential components of the example file, as well as to mark variable components of the pattern design in the so-called *parameters*.

Parameters of all implementations of a pattern are automatically included in the pattern interface. If pattern is defined or explained as a function, then pattern interface is the list of parameters this function will take.



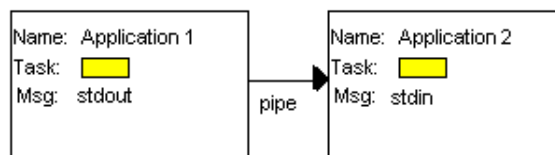
The Pattern Interface

Visual Shell Scripting Tool can be extended in similar fashion by supporting templates for code segments. Any programming language including Unix shell scripting uses reusable components. When a program is initially designed, some of its components can be saved as templates using similar techniques to that of *Pattern by Example*. Then those templates can be put together during the design of the next application.

Those code segments can be simply appended to the script in the linear fashion or use an alternative mechanism, for example, the IC cards. IC Cards were developed by S. K. Chang as a framework for developing interactive multimedia applications. IC (Index Cell) Card is similar to cue card that describes an object in a design.

This card serves as an information template for an object. It features the description of the object, its name, its task, and its interactions with other objects. In the domain of multimedia software design, each IC Card represents a multimedia object. In the domain of patterns, each IC Card can represent a class or an object.

An example of using IC Cards in the pattern design can be illustrated by presenting Unix pipes in terms of IC Card interactions. Diagram below is shows two applications communicating using Unix pipes.



Similar techniques could be used to show interaction between predefined templates in VisualDesktop.

Adding template support to VisualDesktop can significantly enhance its functionality and feature set. This tool has a great potential and can find its niche in the software development as well as an aid to casual Unix users.

## 6. Conclusion

The system described above allows users to write simple Unix shell scripts with greater ease. VisualDesktop is designed for an advanced user with some experience with Unix environment and

programming concepts in order to create time efficient method for creating shell scripts. The way this system is implemented, it will not be suitable for creation of complex shell scripts, since it uses only a subset of Bourne Shell. Future work on this project will lead to the development of more programming constructs can be added to add flexibility and robustness to this implementation. With the addition of pattern templates, this tool can become a fully-fledged development environment that could be ported to other script-like languages, for example Perl.

When fully implemented, this program could prove to be not only an invaluable tool for creation of small shell scripts and “one liner” commands for average Unix user, but for creation of complex applications that can be built from reusable components, defined as software patterns or templates by users themselves.

## References:

- [1] Unix Power <http://www.unixpower.org/>
- [2] Fissell, K., Tseytlin, E., Cunningham, D., Carter, C. S., Schneider, W., and Cohen J. D. “Fiswidgets: A graphical computing environment for neuroimaging analysis”. *Neuroinformatics*, Vol 1, No 1, 2003, 111-125.
- [3] Lowell Jay and Ted Burns. *Unix Shell Programming*, 4<sup>th</sup> edition., Wiley, 1997.
- [4] Ellen Siever, *Linux in a nutshell*, 2<sup>nd</sup> editon, O'Reilly, 1999.
- [5] S. K. Chang. *Multimedia Software Engineering*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.
- [6] James Coplien. "Software Design Patterns: Common Questions and Answeres".
- [7] Alexander, Christopher. *The Timeless Way of Building*. New York; Oxford University Press, 1979.
- [8] Delta Software Technology. *Pattern by Example: User manual*.
- [9] Gamma, Erich, R. Helm, R. Johnson, J. Vlissides. "Design Patterns Elements of Reusable Object-Oriented Software", 1994.
- [10] S. K. Chang, "Visual Languages and Programming", in *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons, 1998.
- [11] S. K. Chang, "Towards a Theory of Active Index", *Journal of Visual Languages and Computing*, Vol. 6, No. 1, March 1995, 101-118.
- [12] S. K. Chang, “An Introduction and General Survey of Multimedia Software Engineering”, <http://www.cs.pitt.edu/~chang/231/vl00/t1.htm>